Project description: Foundations and tools for building well-behaved systems

Katajainen, Jyrki; Elmasry, Amr; Elverkilde, Jon Daniel; Jensen, Claus; Rasmussen, Jens; Simonsen, Bo; Yde, Lars; Artho, Cyrille; Francechini, Gianni; Schupp, Sibylle

Publication date:
2008

Document version
Publisher's PDF, also known as Version of record

Project description: Foundations and tools for building well-behaved systems

Institution:	Department of Computing, University of Copenhagen (DIKU)
Project duration:	1.1.2009–31.12.2011
Principal investigator:	Jyrki Katajainen, Assoc. Prof.
Other investigators:	Amr Elmasry, Humboldt Prof., Max-Planck-Institut für Informatik, Germany; and Alexandria University, Egypt Jon Daniel Elverkilde, B. Sc. Claus Jensen, M. Sc. Jens Rasmussen, B. Sc. Bo Simonsen, B. Sc. Lars Yde, M. Sc., Saxo Bank
Academic partners:	Cyrille Artho, Ph. D., National Institute of Advanced Industrial Science and Technology, Japan Gianni Franceschini, Ph. D., University of Pisa, Italy Sibylle Schupp, Assoc. Prof., Chalmers University of Technology, Sweden

Abstract

We aim at doing basic research on the theoretical foundations on how to build reliable, safe, and fast software systems, and developing tools that make the construction and maintenance of such systems easier. The theoretical questions taken up are related to algorithms, exception safety, and memory management, among other things, and the practical implementation calls for tools to test that components are well-behaved. Our goals are: 1) To study the foundation of a program library in order to gain new knowledge and thereby optimize existing components. 2) To develop software tools which make construction of reliable components easier and are of general interest. 3) To build a program library, the development of which can be used as a reality exercise when training software developers.

1. Background

Programming is hard, and maintenance programming is particularly so since the maintenance programmer has to understand the theory on which the program was built[1]. Constructing and maintaining well-behaved software is still harder.

[1] Peter Naur, Programming as theory building, *Microprocessing and Microprogramming* **15**(5) (1985), 253–261.

CPH STL Report 2008-5, August 2008.

In **performance engineering** the goal is to translate algorithms into efficient computer programs. The performance engineering laboratory[2] (PE-lab) in the Department of Computing[3] at the University of Copenhagen was founded in 1999. One of the major initiatives of the PE-lab is the CPH STL[4] project, which started in 2000 and where the goal is to develop an enhanced edition of the STL, now part of the C++ standard library. Initially, our focus was on time and space efficiency, but now also safety, reliability, and usability of the library components.

The mission of the PE-lab is to educate elite programmers and to do high-quality research related to all aspects of programming, performance programming in particular. Since the foundation of the laboratory one doctor and 21 masters have completed their studies under the supervision of the principal investigator.

In the PE-lab, we conduct basic research; we have several application domains in mind, but we seldom give attention to any specific applications. For example, the C++ standard library, which includes the STL, is claimed to be the most used program library in the world. That is, our work has important implications for the whole society, even if we focus on the computational foundations and, in our research on software tools, on the development of proof-of-concept tools and prototypes, not end-user products.

The actual research is carried out in collaboration between the students associated with the PE-lab and other investigators, including the principal investigator. In this project the PE-lab will extend its focus on performance to include reliability, safety, and usability. We will still work in the areas where we have high expertise: experimental algorithmics and software construction. In-depth expertise in other areas will be provided by our academic partners: Cyrille Artho[5] (program analysis and software tools), Gianni Franceschini[6] (theoretical algorithmics), and Sibylle Schupp[7] (software methodologies and systems).

To attract new students to take part in the project, we will regularly offer courses on topics related to software development. At the moment the following courses in our graduate program are taught by the members of the PE-lab (in collaboration with other research groups): requirements development (first time in 2008), software construction (first time in 2008), generic programming and library development (first time in 2006), and performance engineering (first time in 1998).

We apply a major framework grant with a budget of DKK 3.2 million in total (excl. overhead) covering operating expenses, research travel, attendance at conferences, expenses in connection with the hosting of visiting

[2] http://www.diku.dk/research-groups/performance-engineering/
[3] http://www.diku.dk/
[4] The Copenhagen Standard Template Library; for more information, consult the project website at http://cphstl.dk/.
[5] http://staff.aist.go.jp/c.artho/
[6] http://www.di.unipi.it/~francesc/
[7] http://www.cs.chalmers.se/~schupp/

researchers, salary expenses for a student assistant, and salary expenses for a Ph. D. student, to be distributed over the years 2009, 2010, and 2011. If it is not possible to get support for a Ph. D. student, our secondary preference is a normal framework grant with a budget of DKK 1.5 million in total (excl. overhead). The budget details can be found elsewhere in the application.

Planned research and development is detailed in §§ 2 and 3. Two of the problems we have determined to tackle are discussed in § 2 and the tools we have planned to develop are discussed in § 3.

2. Foundations

In this section we mention open questions which we have encountered in our earlier research. The problems mentioned should give a flavour of the research planned for the next three years.

Safe standard-library components

In the CPH STL project our goal is develop realizations of the C++ standard-library containers that are safer and more reliable than any of the existing realizations. We focus on the following requirements:

Running time: Every container operation must be as efficient as specified in the C++ standard[8], or faster, not only in the amortized sense but in the worst-case sense.

Space usage: The amount of space used must be linear on the number of elements currently stored.

Strong exception safety: Every container operation must be strongly exception safe. That is, each operation completes successfully, or throws an exception and makes no changes to the manipulated container and leaks no resources[9].

Referential integrity: References and iterators to elements stored must be kept valid at all times (except when an element is erased).

The challenge is to achieve the requirements without loss of time efficiency. Also, it is important to understand the performance implications of the requirements compared to other related requirements (robust iterators, persistence, and other forms of exception safety). Of these requirements, the strong guarantee of exception safety has turned out to be difficult to achieve, even though in principle there is no hindrance to it[10].

---

[8] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).

[9] For further details on exception safety, see, for example, Appendix E of the book by Bjarne Stroustrup, *The C++ Programming Language*, Special Edition, Addison Wesley Longman, Inc. (2000).

[10] Jyrki Katajainen, Making operations on standard-library containers strongly exception safe, CPH STL Report **2007-5**, Department of Computing, University of Copenhagen (2007).

Yet another requirement, particularly relevant when programming computers with multi-core and many-core architectures, is:

Thread safety: A piece of program is thread safe if it functions correctly even when it is executed concurrently by multiple threads.

Kasper Egdø faced the challenge of multi-threaded programming when he built a complete transaction system above the C++ programming language and thread-safe standard-library containers on top of that system[11]. It would be important to continue this line of research.

Sound model of computation

When writing a program, the programmer has an abstract machine model in mind, called the random-access machine, the C++ machine, or the Java virtual machine depending on the context. The model is not exact, so the performance of some of the features provided by modern programming languages are not 100% predictable. Predictability is important in several application areas like embedded systems and real-time systems.

It is unclear how the following features can be supported in a predictable manner, if feasible at all:

Random access: In the classical random-access machine the memory is flat and each memory cell is expected to be reachable at unit cost. In reality, the computers have a hierarchy of memory levels and access time depends on the caching strategies used when moving data between different memory levels. The number of operations performed is seldom a predictable measure for the actual running time.

Memory management: In the standard model of memory allocation a memory manager allows one to allocate and free variable-sized segments of memory. Recently, it has been shown that both of these operations can be supported in constant worst-case time[12]. However, this cannot be achieved without excessive memory fragmentation. It is known that, if the total number of memory cells to be allocated is N, any memory manager needs $\Omega(N \lg N)$ cells to serve the requests in the worst case. Under this abstraction, the amount of memory used is not predictable.

Exception handling: When an exception is thrown, the time needed to handle it depends on the distance between the throw point and the catch point (measured in function calls) and the number of objects needed to be destroyed on the way. The running time is not necessarily a constant and it can be difficult to predict[13].

[11] Kasper Egdø, A software transactional memory library for C++, Master's Thesis, Department of Computing, University of Copenhagen (2008).

[12] Gerth Stølting Brodal, Erik D. Demaine, and J. Ian Munro, Fast allocation and deallocation with an improved buddy system, *Acta Informatica* **41**(4-5) (2005), 273–291.

[13] This problem was communicated to us by Bjarne Stroustrup; it appears in print, for example, in Bjarne Stroustrup, Abstraction and the C++ machine model, *Revised Selected Papers from the 1st International Conference on Embedded Software and Systems*, Lecture Notes in Computer Science **3605**, Springer-Verlag (2005), 1–13.

We have attacked the first issue in our earlier research. For example, we introduced the least-recently-used cache model in 1999 as a model of computation for developing algorithms in a hierarchical memory environment[14]; the ideal cache model, which is in common use, was introduced simultaneously and independently with our work[15]. Focus in our forthcoming research will be on the last two issues where interesting work is to be done.

3. Tools

In this section we mention some development challenges we have decided to tackle. The tools to be developed are thighly connected to the CPH STL project, but we expect them to be of interest in other environments as well. The three activities mentioned below are somewhat interconnected. They are expected to form the kernel of the Ph. D. work for which we apply for support. The first (a tool for model-based testing) and second activity (fault injection) could be done independently, at least initially. However, the third activity (testing for thread safety) requires that the first one has already produced a usable tool, although not all features are needed.

Model-based testing for C++

Model-based testing takes a high-level description of the behaviour of a system under test (SUT). Using this model, test cases can be generated automatically. Recently, tools have become powerful enough to make model-based testing applicable in practice. Still, some aspects of currently available tools leave room for improvement. In the most popular openly available tool, ModelJUnit[16], the description of a state machine is written in the target programming language. While this integrates the actions of the SUT with the model, it makes model creation and maintenance unnecessarily complex.

A separation of the underlying state machine in the model from the implementation of its actions promises to deliver a more concise model. In addition to that, existing state-machine visualization tools could then be used directly. Finally, the tool should be flexible enough to handle both C++ and Java. As an extension to existing tools, specification of exceptional behaviour should also be taken into account, accounting for both expected (deterministic) and intermittent (non-deterministic) exceptions.

---

[14] Jesper Bojesen, Jyrki Katajainen, and Maz Spork, Performance engineering case study: Heap construction, *Proceedings of the 3rd International Workshop on Algorithm Engineering*, Lecture Notes in Computer Science **1668**, Springer-Verlag (1999), 301–315.

[15] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran, Cache-oblivious algorithms, *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society (1999), 285–297.

[16] Mark Utting and Bruno Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann (2007).

Fault injection and exception safety

Fault injection simulates exception occurrence in software. Exceptions often occur due to underlying hardware or network failure, but also on other unexpected events, such as unavailability of memory. Many kinds of exceptions, such as input/output errors, occur sporadically and cannot be reproduced in a deterministic way. This makes it hard to test exception handlers properly.

Fault injection has been devised to deal with this problem[17]. However, fault injection is often performed on a black-box level, without knowledge of the implementation or test structure. Randomized fault injection is often used, making verification of exception handlers unreliable.

When applied to unit tests, fault injection can target individual failures by re-using a particular unit test. Compared to random testing, performance is improved by an order of magnitude[18].

Such a fault injection technique could also be used to test C++ library components for different levels of exception safety. If test cases are generated from a formal model, the initial coverage analysis phase used in previous work[18] may even be subsumed by information contained in the model.

Expedient testing for thread safety using model-based testing

Many libraries promise the thread safety of their implementation. However, a correct implementation is difficult to achieve in practice. The problem is that the outcome of a particular execution depends on the execution schedule, which varies between test runs and cannot be controlled directly.

Various techniques have been derived to improve detection of concurrency-related problems. The two most important techniques include controlled schedule perturbation, to cover a larger subset of the potential behaviours[19], and data race detection. In the latter approach, potentials for access conflicts are detected by observing locking patterns. This can reveal access conflicts even if no incorrect result is produced by a test execution[20].

Modern software often has a rich API, allowing for different ways to reach a particular goal. Model-based testing makes it possible to execute several

[17] M. Hsueh, T. Tsai, and R. Iyer, Fault injection techniques and tools, *IEEE Computer* **30**(4) (1997), 75–82.

[18] Cyrille Artho, Armin Biere, and Shinichi Honiden, Exhaustive testing of exception handlers with Enforcer, *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects*, Lecture Notes in Computer Science **4709**, Springer-Verlag (2007), 26–46.

[19] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur, Framework for testing multi-threaded Java programs, *Concurrency and Computation: Practice and Experience* **15**(3-5) (2003), 485–499.

[20] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, Eraser: A dynamic data race detector for multithreaded programs, *ACM Transactions on Computer Systems* **15**(4) (1997), 391–411.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran, Goldilocks: Efficiently computing the happens-before relation using locksets, *Proceedings of the 1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification*, Lecture Notes in Computer Science **4262**, Springer-Verlag (2006), 193–208.

alternative behaviours. This technique could be extended to generate scenarios using concurrent access to a data structure. Existing verification techniques for concurrent software could then be applied to a large number of scenarios, making detection of subtle problems more likely.

On behalf of the research group

Copenhagen, 29 August 2008

Jyrki Katajainen
Assoc. Prof., Ph. D.